Progress Thread Placement for Overlapping MPI Non-Blocking Collectives using Simultaneous Multi-Threading

Alexandre Denis[1], Julien Jaeger[2], and Hugo Taboada[1,2]✉

[1] Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP, France
{alexandre.denis,hugo.taboada}@inria.fr
[2] CEA, DAM, DIF, F-91297 Arpajon, France
{julien.jaeger,hugo.taboada}@cea.fr

Abstract. Non-blocking collectives have been proposed so as to allow communications to be overlapped with computation in order to amortize the cost of MPI collective operations. To obtain a good overlap ratio, communications and computation have to run in parallel. To achieve this, different hardware and software techniques exists. Dedicated some cores to run progress threads is one of them. However, some CPUs provide Simultaneous Multi-Threading, which is the ability for a core to have multiple hardware threads running simultaneously, sharing the same arithmetic units. Our idea is to use them to run progress threads to avoid dedicated cores allocation. We have run benchmarks on Haswell processors, using its Hyper-Threading capability, and get good results for both performance and overlap only when inter-node communications are used by MPI processes. However, we also show that enabling Simultaneous Multi-Threading for intra-communications leads to bad performances due to cache effects.

1 Introduction

MPI is the standard interface for communications in HPC applications. It is used by applications for inter-node (i.e. network) and intra-node (processes on the same node) communications. The cost of communications is one of the main obstacles to get a good speedup for parallel applications. To amortize the cost of MPI communications, application programmers try to overlap communications with computation by using non-blocking communication primitives, and let them progress in background while keeping the CPU busy with computation.

Initially the non-blocking communications were only available for point-to-point communications. The extension of the non-blocking communications to collective operations (i.e. primitives that involve more than two nodes, such as broadcast, reduce, scatter, gather, ...) is an addition of the latest major MPI version [1]. It opens the door to computation/communication overlap for collective operations too. However, collective communications are more CPU-hungry than point-to-point communications, as they have to handle the collective algorithms,

and even some computations for reduction collectives. Therefore, it is harder to make them progress in background.

Most processors nowadays include *Simultaneous Multi-Threading* [2] (SMT, commercially known as *Hyper-Threading* on *Intel* processors), which is the ability for a core to have multiple *hardware threads* running simultaneously, sharing the same arithmetic units. A lot of scientific applications don't use all hardware threads, leaving them idle. Thus it seems like a natural idea to use these idle *hardware threads* to make communication progress. Since communication typically doesn't use arithmetic units, it is expected that placing progress threads on *hardware threads* will bring background progression for free. We distinguish the case of *network* (inter-node) communication, where the progression thread merely execute the algorithm for the collective operation, the rendez-vous protocol, programs DMA on the NIC, but overall doesn't burn a lot of CPU cycles; and the case of *shared-memory* (intra-node) communication, where the transfer is essentially a `memcpy`, which may be heavier on the CPU.

This paper focuses on what happens when placing MPI non-blocking collective progress threads on *hardware threads*. We show that using *SMT* for network communications leads to good results for both performance and progression. We also show that using *SMT* for intra-node (shared memory) communications leads to bad performances due to cache effects.

The rest of the paper is organized as follows. Section 2 presents related work about computation/communication overlap in general, and for collective communication in particular. Section 3 describes how communication progression works inside the MPC framework. Section 4 presents progress threads placement for inter-node and intra-node communications and results on Haswell processors, using Hyper-Threading.Then, Section 5 explains how intra-node communications can interfere on the computation when Hyper-Threading are used to make communication progression, before concluding in Section 6.

2 Related Works

The topic of communication progression has already been studied for some aspects in the literature. Several strategies do exist for background progression of point-to-point communications, such as offloading the communication to hardware [3, 4] and let the hardware do the progression; use of a thread [5] or process [6] dedicated to communication progression; opportunistic scheduling of communication tasks [7, 8].

MPI non-blocking collective communications are more difficult to make progress in the background, since not only the data transfer but the collective algorithm too needs to progress, which makes it harder to rely on hardware. There is specific work [9] for hardware-assisted progression on Blue Gene, or offloading shared memory collectives to a kernel module [10] (although authors only address performance of blocking collectives, not progression of non-blocking collectives). The reference NBC implementation [11] relies on a progress thread,

with some tricks [12] to improve overlap on InfiniBand, but without any study about the impact of progress thread placement.

Hyper-Threading usage for non-blocking operations progression has already been studied in [13], with the use of `MONITOR`/`MWAIT` instructions on progress threads in order to avoid resource contention with the computational thread on the same physical core using another Hyper-Thread on process based MPI (network communication on intra node). However, `MONITOR`/`MWAIT` being privileged instructions usable only from kernel, this approach may not be used broadly on production clusters. Moreover, these instructions are inherently slow, which reserve them for coarse grain cases. Our approach is different because we rely only on bare Hyper-Threading accessible from user-space, and study different placements for both process based MPI (intra-node communications on network) and thread based MPI (intra-node communication with `memcpy`).

3 Non-blocking collective progression inside the MPC Framework

The MPC [14] framework provides implementations for several parallel programming languages, such as MPI, OpenMP or POSIX threads. MPC provides two flavors for MPI: a process-based implementation and a thread-based implementation. Moreover, MPC also provides its own user thread scheduler. This scheduler handles the threads of all programming languages implemented in MPC, or build on top of the POSIX threads implementation provided by MPC, and allows to bypass the system scheduler.

MPC uses a tuned version of libNBC [11] to implement MPI 3 Non-Blocking Collectives. One progress thread is created for each MPI process. Thus, with the thread-based version of MPI, the MPC scheduler has the knowledge of all MPI processes and all progress threads present on a node. This knowledge allows to easily implement different placement algorithms for all these threads. The default behavior is for MPI "thread" to be bound with a scatter policy, and their corresponding progress threads to be bound to the closest idle cores (or to the same core if no idle cores are available).

In this implementation, a MPI non-blocking collective is decomposed in MPI point-to-point non-blocking calls fulfilling the collective algorithm. When a MPI non-blocking collective is called, each MPI process creates a *schedule* containing requests for the point-to point non-blocking calls corresponding to its part of the collective algorithm, and attach it to its associated progress thread. Thus, the progress threads handle the communication described by the schedules while MPI processes continue to execute computation. However, MPC has a non-preemptive scheduler, thus it is not able to make communication progress on the same core as the application with a seamless interleaved scheduling. A solution is to dedicate some cores to communication progression. In this paper we investigate the use of hardware threads instead of full cores for communication progression.

4 Progress Threads Placement for MPI Communications on Hyper-Threads

In this Section, we benchmark various placement schemes for placement of progress threads, using SMT or not, for network communications and shared-memory communications.

We will use Haswell processors featuring *Hyper-Threading*, the incarnation of *Simultaneous Multi-Threading* in Intel processors. It consists in allowing execution of two different threads (or more depending on the architecture) at the same time on a single core. Generally, applications do not use Hyper-Threading to perform more computation because it leads to Floating-Point Unit (FPU) contention. However, progress threads do not need the FPU to make communication progression, or scarcely for floating-point reduction operations. Thus, progress thread placement using Hyper-Threading seems to be a good idea.

After describing our experimental setup, we present results and observations on the use of Hyper-Threading to perform communications for the two distinct cases: pure network communications, and pure intra-node communications.

4.1 Benchmark

We implemented our own micro-benchmarking tool to evaluate the performance of different progress threads placement. This tool performs a non-blocking collective communication overlapped with a matrix-matrix multiply. It works similarly to the Intel MPI Benchmarks [15] except that the problem size is fixed, allowing us to have the same computation workload for the different progress threads placement. We arbitrary set the buffer size to 2 MB and sized the computation workload to reach perfect overlap when we have progress threads dedicated cores.

We ran our benchmark on a many-core architectures: an Intel Xeon E5-2698 v3 @2.30GHz with 32 cores per node, and 128 GB of RAM (Haswell).

While our placement policy for MPI processes stays the same (scatter policy), we test three different progress threads placement configurations:

- "dedicated-core": each progress thread is bound on another dedicated core. We use twice more cores than both the other cases.
- "no-smt-bind": the progress threads are bound on the MPI process core and Hyper-Threading is disabled.
- "smt": each progress thread is bound on its MPI process core but on another Hyper-Thread.

For each configuration, we measure the time of the computation (t_{cpu}), the communication time (t_{comm}) and the total execution time (t_{ovrl}), all times measured when overlapping communication with computation. We get t_{ovrl} close to the maximum of t_{cpu} and t_{comm} in case of good overlap; it is closer to the sum in case operations get serialized. Please note that t_{comm} and t_{cpu} may vary depending on threads placement if computation slows down communication or

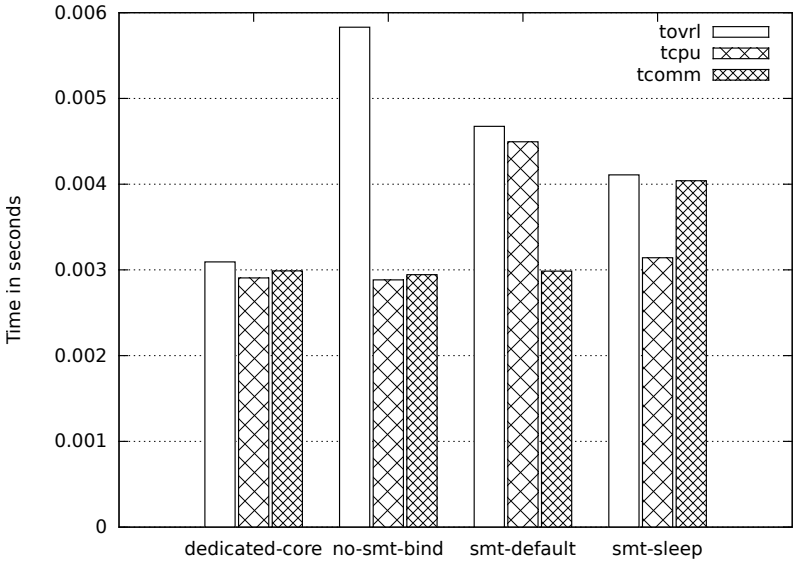

Fig. 1. Result of dedicated-core, no-smt-bind, smt-default and smt-sleep for `Ialltoall` operation with constant-size buffer of 2 MB on 8 nodes with 8 MPI processes.

if computation slows down communication when both are run at the same time. We use the same overlap definition as the Intel MPI Benchmarks [15]:

$$overlap\_ratio = 100 * \frac{max(0, min(1, (t_{comm} + t_{cpu} - t_{ovrl})}{min(t_{comm}, t_{cpu}))}$$

4.2 Inter-node Communications on Hyper-Threads

To study the impact of using hyper-threads only for inter-node communications, we ran our benchmark on 8 Haswell nodes, with only one MPI process per node. This is a usual configuration when MPI is combined with a threaded programming model (e.g. OpenMP) handling intra-node communications.

The results for inter-node communications are depicted in Figure 1. The best results are obtained for the "dedicated core" placement, with an overlap ratio of 96% for an execution time of 3.0 ms. This is the expected behavior since a dedicated core for each progress thread makes the communication progress run smoothly in background, leading to an almost perfect overlap. However, this configuration uses twice as many cores as the other cases.

For the "no-smt-bind" placement, no overlap happens and the execution time doubles (5.8 ms). This is the expected behavior since MPC being non-preemptive, computation and communication end up serialized if computation thread and progress thread are placed on the same core. We observe that communications need some CPU resources to progress, not necessarily for the network itself,

but at least to execute the algorithm of the collective and for the rendez-vous protocol for large messages.

The "smt" placement with default settings leads to an overlap ratio of 94% for an execution time of 4.6 ms. While the overlap ratio is good, we also observe that the t_{cpu} increases significantly. This is due to our MPI implementation. When Hyper-Threading is enabled, MPC creates an OS thread per logical core (Hyper-Thread). By default, this thread is populated with an idle user thread, spending its time busy waiting for work. As nothing is planned for this thread, it will permanently hinder the CPU with its busy waiting, thus slowing down the computation done on the other *hardware thread* sharing the same core.

To assess this behavior, we inserted a `usleep` call ($2\,\mu s$) to diminish the impact of this busy waiting in the idle thread.

With this version, called smt-sleep in Figure 1, we observe an improvement of t_{ovrl} by a factor of 1.42 over the default MPC configuration (no-smt-bind) and an overlap ratio of 98%. In this version, t_{cpu} is only marginally impacted, which shows this tuning successfully mitigates contention between communication and communication. Since the idle thread is sleeping most of the time, the computation thread is indeed not hindered and the computation time is back to normal. However, when progression happens, the sleep calls reduce progression performance and the communication time is higher. Hence, it is possible to find a trade off to get best of both worlds.

As a summary, placing progress thread on hyper-threads improves both execution time performance and overlap ratio for network inter-node communications. It alleviates the need for dedicated cores for communication progression.

4.3 Intra-Node Communications on Hyper-Threads

The common way to achieve intra-node communications is to copy a buffer from the source to the destination. For process based MPI implementation, such as Open MPI, MPICH, MVAPICH, Intel MPI or NewMadeleine, this can be performed using a shared memory segment across all the processes in the node. This technique allows MPI ranks to copy the buffer directly in the shared memory segment.

In the MPC framework, with the thread-based flavor, all MPI ranks are threads. This implies that the whole memory is shared in the same address space. Copies of buffers can be performed directly with a single *memcpy* call.

We ran our benchmark on one single Haswell node, with one MPI rank per core. We test two different thread placement configurations: the "no-smt-bind and the "smt" placement described in the Section 4.1. For each configurations, we measure the computation time (t_{cpu}), the communication time (t_{comm}) and the total execution time (t_{ovrl}) when overlapping communications with computation.

The results are depicted in Figure 2. For both "no-smt-bind" and "smt" placement, we observe that $t_{ovrl} = t_{cpu} + t_{comm}$, which means no overlap happens. We also observe a 44% increase of the total time t_{ovrl} when placing progress

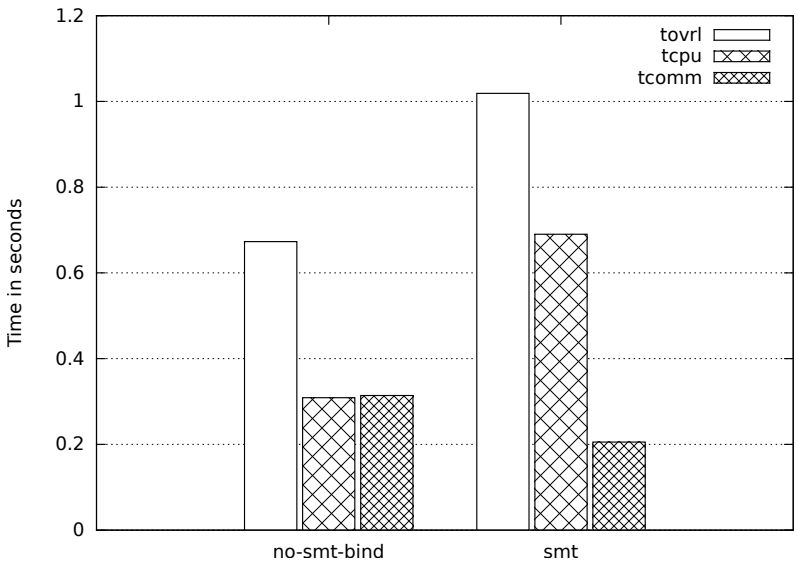

Fig. 2. Result of no-smt-bind and smt for `Ialltoall` operation with constant-size buffer of 2 MB on 1 nodes with 32 MPI processes.

threads on hyper-threads, due to the huge increase of computation time. This is a completely different behavior than with inter-node communication.

From this observation, it is clear that placing progress threads on hyper-threads has a huge impact on computation performance when communications take place in shared memory. We investigate this issue in the Section 5.

5 Cache Effects with Hyper-Threading

In this Section, we investigate how a communication thread on a hyper-thread negatively impacts the computation performance on the same core. We focus on cache effects caused by multiple hardware threads on the same core competing for cache lines, an effect known as *cache thrashing*.

We implemented a micro-benchmark to confirm our assumptions that cache effects occur when Hyper-Threading is used to perform the progression of intra-node communications. The benchmark runs a 1024×1024 matrix multiplication in a thread bound to a single core; we call it the *computation thread*. Another thread is created to simulate the progression of intra-node communications by performing a `memcpy` call in a loop; we call it the *memcpy thread*. We focus on the impact of this thread on the *computation thread*.

We test three different threads placement configurations :

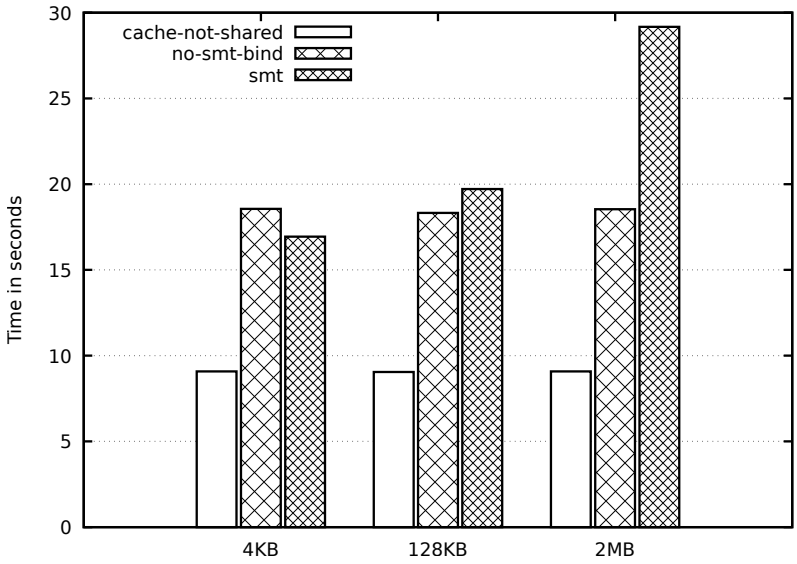

Fig. 3. Result of Time dgemm for no-smt-bind and smt configurations for 4 KB, 128 KB and 2 MB on a 32 core Haswell processor.

- "cache-not-shared": the *computation thread* is bound on a single core and the *memcpy thread* is bound on another socket. Threads do not share any cache.
- "no-smt-bind": the *computation thread* is bound on a single core and the *memcpy thread* is bound on the same core. Hyper-Threading is disabled.
- "smt" the *computation thread* is bound on a single core and the *memcpy thread* is bound on the same core but on the other Hyper-Thread.

For each configuration, we run our tests with three buffer sizes for the *memcpy thread*, 4 KB, 128 KB and 2 MB on a dual socket Haswell processor, with 16 cores per socket and 2 Hyper-Thread per core.

We measure the time of the computation for these three different threads placements with different buffer sizes. We observe in Figure 3 that for all buffer sizes, we obtain a 9 seconds execution time with the "cache-not-shared" placement. This time doubles when we use the "no-smt-bind". The reason is that the first two placements do not compete for the caches. In the first case, the two threads are not located on the same socket. In the second case, the two threads are located on the same core, but without Hyper-Threading, execution is interleaved. Hence, when the *computation thread* runs, the *memcpy thread* is paused, and, after context switching between these threads, the *memcpy thread* runs while the *computation thread* is paused. If data may be removed from the caches after context switching, no competition for the cache occurs while a thread is running between context switches. However, for the "no-smt-bind" placement,

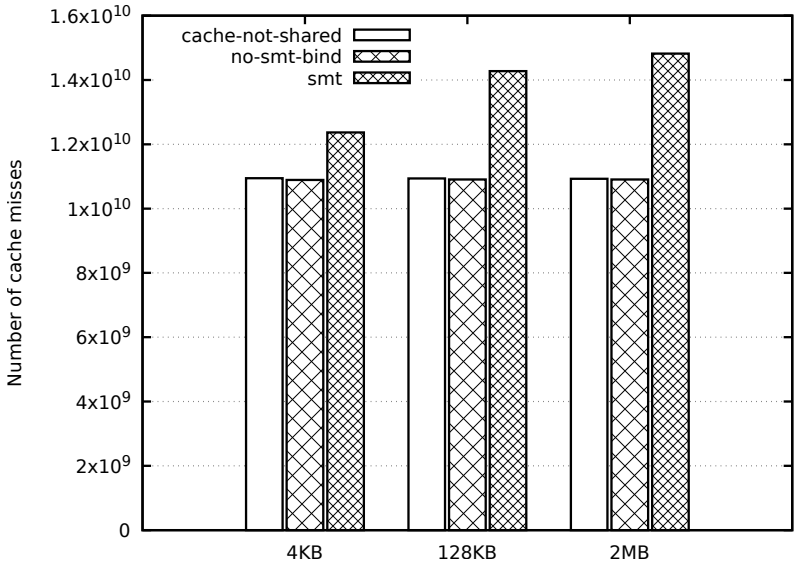

Fig. 4. Result of cache L1 miss for cache-not-shared, no-smt-bind and smt configurations for 4 KB, 128 KB and 2 MB on a 32 core Haswell processor.

the two threads share the same core without Hyper-Threading. Thus, all the workload from both threads are running on the same resources. Since we fixed the *memcpy thread* to run as long as the *computation thread*, it doubles the workload per core, hence also doubling the execution time.

For the "smt" placement, the *memcpy thread* is bound to the same core as the *computation thread* but on another Hyper-Thread. We observe a different behavior. Execution time is slower when the buffer size increases whereas the execution time remains constant between buffer sizes for the other placements. The computation is slower when the *memcpy thread* manipulates a larger amount of data: it is a typical symptom of *cache thrashing*.

To assess this hypothesis, we use the Performance Application Programming Interface (PAPI) [16] to collect the L1 cache misses. We see in the Figure 4 that the number of cache misses is constant between both "no-smt-bind" and "cache-not-shared" placements. This is expected because the *computation thread* and the *memcpy thread* do not share the caches for the "cache-not-shared" placement. For the "no-smt-bind" placement, these threads are scheduled one after the other and no additional cache misses occurs.

For the "smt" placement, we observe additional cache misses compare to the two previous placements. This is due to Hyper-Threading being enabled. Both threads are executed on the same core simultaneously sharing the caches. Both of them needs to fetch their cache lines to execute their jobs. Contention happens

and leads to additional cache misses because the *memcpy thread* evicts cache lines of the *computation thread*.

It is now common to use non-temporal memory operations for shared memory operations in MPI libraries. The non-temporal memory copy, introduced with SSE2 instruction set, do not store in cache data sent to memory (i.e. it forces a *write around* cache policy). However, only the *write* operation bypasses the cache, not the *read*. Benchmarks with non-temporal memory copy exhibits the same results as with regular memory copy.

These results demonstrate that using Hyper-Threading for communication progression in shared-memory causes a flood of cache misses, which severely degrades the performance of computation on the same core.

6 Conclusion and Future Work

Overlapping communications with computation is the key to amortize the cost of communications, especially for collective communications which are heavier than point-to-point communications. Approaches for progression relying on a progress thread per MPI rank may suffer from competition between communication and computation.

In this paper, we have studied the placement of progress threads for MPI non-blocking collective on hyper-threads and compared it against dedicated cores. We have brought a comprehensive benchmark and full performance analysis of using hyper-threads for communication progression on Haswell processor.

We have tested several progress thread placements and obtained an overlap ratio of 98% of *network* communications when placing progress threads on hyper-threads. We have shown that this scheme leads to performance degradation for *shared memory* communication, and highlighted its cause in cache thrashing.

As a consequence of this work, the optimal placement for a network communication and a shared-memory communication is not the same, which is not achievable through the use of a single progress thread making progress for all communications. As future works, we plan to have communication progression rely on *tasks* rather than on a thread, which will allow for a greater flexibility in placement.

References

1. MPI Forum: MPI: A Message-Passing Interface Standard Version 3.0 (September 2012)
2. Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., Tullsen, D.M.: Simultaneous multithreading: a platform for next-generation processors. IEEE Micro **17**(5) (Sept 1997) 12–19
3. Sur, S., Jin, H., Chai, L., Panda, D.: RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM New York, NY, USA (2006) 32–39

4. Rashti, M.J., Afsahi, A.: Improving communication progress and overlap in MPI Rendezvous protocol over RDMA-enabled interconnects. In: High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on, IEEE (2008) 95–101
5. Hoefler, T., Lumsdaine, A.: Message Progression in Parallel Computing - To Thread or not to Thread? In: Proceedings of the 2008 IEEE International Conference on Cluster Computing, IEEE Computer Society (Oct. 2008)
6. Lai, P., Balaji, P., Thakur, R., Panda, D.: ProOnE: A General Purpose Protocol Onload Engine for Multi- and Many-Core Architectures. (June 2009)
7. Denis, A.: pioman: a pthread-based Multithreaded Communication Engine. In: Euromicro International Conference on Parallel, Distributed and Network-based Processing, Turku, Finland (March 2015)
8. Si, M., Peña, A., Balaji, P., Takagi, M., Ishikawa, Y.: MT-MPI: multithreaded MPI for many-core environments. In: Proceedings of the International Conference on Supercomputing. (06 2014)
9. Almási, G., Heidelberger, P., Archer, C.J., Martorell, X., Erway, C.C., Moreira, J.E., Steinmacher-Burow, B., Zheng, Y.: Optimization of MPI Collective Communication on BlueGene/L Systems. In: Proceedings of the 19th Annual International Conference on Supercomputing. ICS '05, New York, NY, USA, ACM (2005) 253–262
10. Ma, T., Bosilca, G., Bouteiller, A., Goglin, B., Squyres, J.M., Dongarra, J.J.: Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs. In IEEE, ed.: 40th International Conference on Parallel Processing (ICPP-2011), Taipei, Taiwan (September 2011)
11. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07, IEEE Computer Society/ACM (Nov. 2007)
12. Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC'08 Workshop. (Apr. 2008)
13. Miwa, M., Nakashima, K.: Progression of MPI Non-blocking Collective Operations Using Hyper-Threading. (03 2015) 163–171
14. Pérache, M., Jourdren, H., Namyst, R.: MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In Springer, ed.: the 14th International Euro-Par Conference. Volume 5168 of LNCS., Las Palmas de Gran Canaria, Spain (August 2008) 78–88
15. : IMB-NBC benchmarks. https://software.intel.com/fr-fr/node/561946 Accessed: 2018-05-10.
16. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting Performance Data with PAPI-C. In Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E., eds.: Tools for High Performance Computing 2009, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 157–173